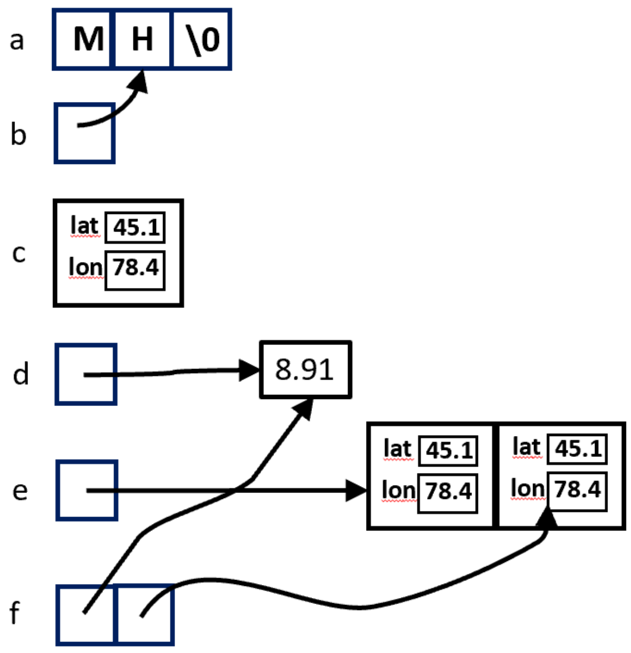
CPSC 223 - Fall 2018 - Exam #1

Write your name and NetID on only this page of this exam package in the boxes provided and **write your answers on the front of the provided sheets no closer than $\frac{1}{2}$ inch from the edges of the pages**.

Name

NetID

Problem 1 (16 points): Write the declarations and initializations to reflect the memory diagram shown below. For this and subsequent diagrams, there should be no local variables other than `a`, `b`, `c`, `d`, `e`, and `f`, everything not labelled with a local variable name is dynamically allocated, the numeric values are `double`s, the character values are `char`s, there is a `struct location` with `typedef`-ed alias `location`, that struct has `double` fields `lat` and `lon`, and you may assume that the appropriate header files have been included.



Problem 2 (4 points): Write the calls to `free` required so that there are no memory leaks after the local variables in the diagram from Problem 1 go out of scope (again, assume that there are no other variables aside from those shown in the diagram).

Problem 3 (14 points): Assuming that local variables `a` through `f` have been set up according to the diagram in Problem 1, indicate for each of the following groups of statements whether one or more of the statements are invalid (will not compile without warnings or errors), or if all are valid then give the output of the `printf` or state that the behavior is undefined (for example, when pointers are used dangerously). Consider the groups of statements in isolation; **do not** consider the effects of previous pairs of statements.

```
a) (*b)++;
   printf("%s\n", a);

b) b--;
   printf("%s\n", b);

c) d = &f;
   printf("%f\n", *d);

d) d = f[1];
   printf("%f\n", *d);

e) f = &d;
   printf("%d\n", *f[0]);

f) c = e[2];
   printf("%f\n", c.lat);

g) d--;
   printf("%f\n", d[1]);
```

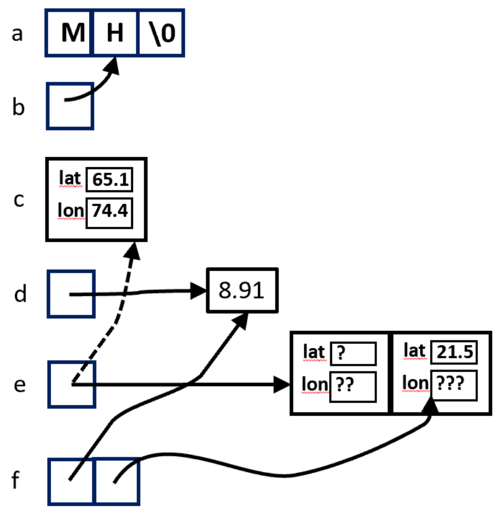
Problem 4 (12 points): Assuming that local variables `a` through `f` have been set up according to the diagram below (with the initial state of the pointers shown with the solid lines), write the statements to

(a) change the 8.91 to the value represented by "???" (whatever it is)

(b) copy the values marked "?" and "??" into `c`

(c) change `e` as indicated by the dashed line

(d) change the original copy of the "???" to 10.0 (assume that the changes from (a)-(c) have taken effect)



Problem 5 (16 points): Modify the following program so that `location` is an opaque struct. Show the changes to `main.c` and `location.h`. **You may omit the changes to `location.c`**, but if you add functions you should describe them briefly in the appropriate place in natural language, leaving out descriptions of error handling. Your finished program should have no memory leaks or other errors that would be reported by valgrind.

```
// --------------------- location.h ---------------------
#ifndef __LOCATION_H__
#define __LOCATION_H__

struct location { double lat; double lon; };

typedef struct location location;

// WRITE ADDITIONS TO location.h ON THE FOLLOWING PAGE

// copies to the struct pointed to by mid the point halfway between from and to
void location_midway(const location *from, const location *to, location *mid);

#endif

// --------------------- location.c ---------------------
#include <math.h>
#include "location.h"

void location_midway(const location *from, const location *to, location *mid) {
  // HEAVY-DUTY MATH OMITTED FOR BREVITY
}

// --------------------- main.c ---------------------
#include <stdio.h>
#include "location.h"

int main() {
  double lat1, lon1, lat2, lon2;
  // INITIALIZATIONS AND ERROR HANDLING AND OMITTED FOR BREVITY

  location   hq1 = { lat1, lon1 };

  location   hq2 = { lat2, lon2 };

  location   mid;

  location_midway( &hq1,  &hq2,  &mid );

  printf( "%f %f\n", mid.lat, mid.lon );








}
```

Write additions to location.h here. Indicate what is removed from location.h on the previous page.

Problem 6 (20 points): The following function reads the given number of `double`s from standard input. For each value it reads, it transforms the value using the `normalize` function and saves the transformed value in the array. The array is returned at the end of the function. Modify the function so that it

(a) performs error checking on the `scanf` call so that it stops reading when it becomes impossible to read a `double` and in all cases returns the number of values it actually read through a new (simulated) reference parameter along with the array containing the values read through the returned pointer;

(b) performs error checking on the memory allocation (returning NULL and 0 if there was an error);

(c) reads from a file whose name is given by an additional string parameter instead of standard input, closing the file any time the function ends, and returning NULL and 0 if there was an error opening the file (and modify whatever error checking you added from (a) to now work with the file); and

(d) transforms the values read by calling a function passed as an additional argument, where `normalize` should be one of the functions possible to pass as that argument (no error checking is necessary on this additional argument).

After all of your changes, the call `vals = read(10, &count, "infile", normalize);` should be legal, assuming that `vals` is declared as a pointer to a `double` and `count` is an `int`.

```
double normalize(double x) {
  x = x - 360.0 * (int)(x / 360.0);
  return x >= 0.0 ? x : x + 360.0;
}

double *read(int   n                                                    )  {




  double *values = malloc(sizeof(double) * n);





  for (int i = 0; i < n; i++)   {
      double x;

      scanf("%lf", &x);





      values[i] = normalize(x);
    }




  return values;
}
```

Problem 7 (18 points):

(a) Complete the following function `concat_all` that takes two arguments: an non-empty array of strings and its size. The function returns a new string containing all the strings in the array concatenated together with single space characters between them. The space allocated to the string should be the smallest possible. For each blank in the function choose one of items A-O to fill the blank and write the corresponding letter. Each item may be used zero, one, or more times. Write up to three statements at the end of the function to complete it. You may not add other code. You may assume that the array passed to `concat_all` is not `NULL`, is non-empty, that none of the strings are `NULL`, that `n` is the correct length of the array, and that the concatenated string is not too long to fit in memory.

```
char *concat_all(_____________arr[], int n) {
  int tot_size = 0;

  // determine the length of the result
  for (int i = 0; i < n; i++)
    {
      tot_size += ______(_______) + _____;
    }
  tot_size += ______ + _______;

  // create an array to hold the result
  _____ result[tot_size];

  // initialize the result
  ______(result, ____);

  // concatenate each string in the array to the end of the result
  for (int i = 0; i < n; i++)
    {
      if (i > 0)
        {
  _______(result, ______);
        }
      ______(result, ______);
    }

    // WRITE UP TO THREE LINES HERE TO FINISH THE FUNCTION CORRECTLY




}
```

```
A: char       D: strlen     G: 0       M: arr         K: ""
B: char *     E: strcat     H: 1       N: arr + i     L: " "
C: char **    F: strcpy     I: -1      O: arr[i]
                            J: n
```

(b) Assuming that there are n strings each of length between n and $2n$, what is the asymptotic (big-O) running time of your completed `concat_all` function in terms of n?